マルチコア用プロセスペアフレームワーク
取扱説明書

2010 月 02 月

第1章　はじめに

1.1　はじめに

本書では、組み込み向けマルチコア上で動作するマルチコア用プロセスペアの導入を支援するソフトウェアフレームワーク（以降、本ＦＷまたは単にＦＷと呼ぶ）について説明します。ＦＷの動作、機能、使用方法（アプリケーションプログラムの開発方法）の順で書かれています。また、ソフトウェアフォルトトレランス技術や、典型技法の一つであるプロセスペアの知識が多少あることが望ましいです。

1.2　開発環境

本ＦＷおよびサンプルプログラムはＣ言語を使用し、統合開発環境ＨＥＷを用いて開発されています。ＦＷに対応する開発環境、コンパイラのバージョン次のようになります。

開発環境	バージョン	コンパイラ	バージョン
ＨＥＷ	Ｖ 3.01.08.000 以降	ＳＨＣ (ルネサステクノロジ)	V9.0.1.0 以降

また、現在ＦＷが対応している開発チップは次に示す通りです。

対応チップ	概要
AP-SH2AD-0A (ルネサス製)	デュアルコア CPU、SH2A-DUAL(SH7205) を搭載

第2章　フレームワークについて

2.1　フレームワークの概要

本ＦＷは、ソフトウェアフォルトトレランス技術の一つであるプロセスペアを、組み込み向けマルチコアチップに実装し、そのフォルトトレランス（耐障害性）を実現するためのソフトウェア構造の導入を支援するものです。

2.2　フレームワークの動作

本ＦＷが導入を支援するマルチコア用プロセスペアと、ＦＷを適用する事で得られる動作を説明します。

2.2.1　マルチコア用プロセスペアについて

プロセスペアとは、マルチＣＰＵ環境において、同じソフトウェアを異なったＣＰＵに実装し、正常時はメイン側のＣＰＵのみが稼働します。何らかの障害が発生した時は、もう一方のサブＣＰＵ側のソフトウェアを稼働させ、サービスを継続させるものです。マルチコア用プロセスペアとは、このプロセスペアの動作をマルチコア上で実現するものです。本ＦＷは、そのために必要となるソフトウェア構造作成を支援します。

2.2.2　動作

本ＦＷを適用した場合に得られる動作について図2.1を用いて説明します。アプリケーションプログラム（以降ＡＰＰと略記します）や、ＦＷの利用法によって詳細な動作は異なってきますが、ここでは典型的な動作について説明します。ＡＰＰは一方のＣＰＵのみで稼働します（最初はメイン側、障害発生後はサブ側）。稼働中のＣＰＵはＡＰＰに加え、自ＣＰＵに対してエラーチェックを行います。エラーチェックでエラーが無いと判断された場合は、その時点のチェックポイント情報（ＡＰＰを再開させるための情報）を共有メモリに保存します（この動作をチェックポインティングと呼びます）。もし、エラーが検出された場合は、待機中のサブＣＰＵが稼働し、チェックポイント情報を利用しＡＰＰを再開します。この

動作が基本となりますが、エラーチェックを行うタイミングや、チェックポイント情報を残すタイミングなどはＦＷ利用者が自由に選択を行うことが可能な構造となっています（詳細については４章で説明します）。また、必要に応じて、一度障害が発生したＣＰＵをソフトウェアリセットし、再び障害の発生に備える、多重ＣＰＵ切替機能（詳細５章）を選択する事も可能です。

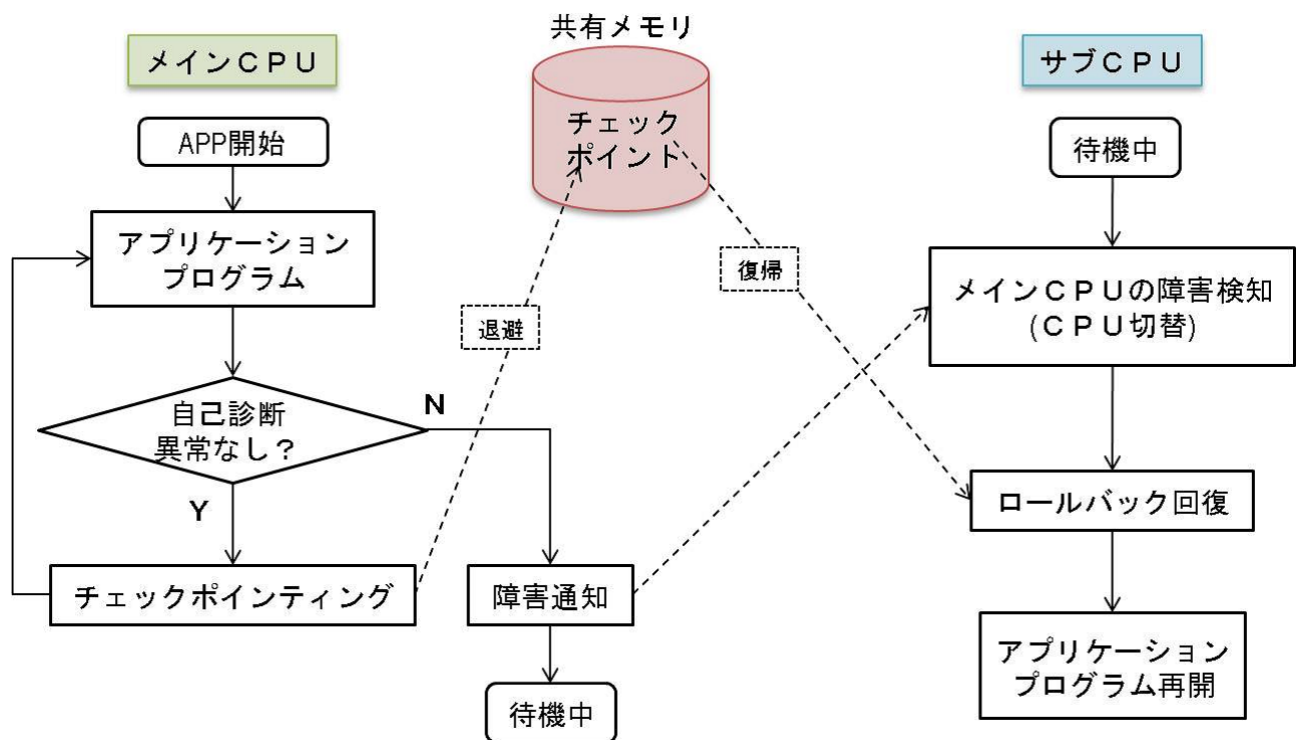



図 2.1：動作図

2.3 フレームワークのファイル構成

ＦW用のファイル

- FW_main.c(&h) ：ＦWのメイン処理
- FW_wdt.c(&h) ：ウォッチドッグタイマの処理
- FW_recover.c(&h) ：システム回復処理
- FW_config.h ：ＦW設定ファイル
- FW_vector.c(&h) ：割り込みベクタテーブル
- AP_SetUp.c(&h) ：ＣＰＵの初期化とＡＰＰ依存の設定を行う
- AP_main.c(&h) ：ＡＰＰのメイン処理
- ckp_table.h ：チェックポイント関数テーブル
- IH_Template ：割り込みハンドラ用テンプレートファイル
- make_ckp_table.rb ：自動化スクリプト（詳細は後述
（開発チップに依存するファイル）
- 7205.h ：SH7205 内部レジスタ定義ファイル
- BoradDepend.h ：ボード依存定義ファイル

サンプルプログラム用のファイル

- main.c ：メイン処理
- sci.c ：シリアル割込み処理 (ハンドラファイル)
- timer.c ：コンペアマッチタイマ（以降ＣＭＴと記述）割込み処理 (ハンドラファイル)
- common.h ：ＡＰＰ共通ヘッダファイル

2.4　サンプルプログラムについて

サンプルプログラムについて説明します。本ＦＷに付属のサンプルプログラム（以降、サンプルと略記）は、ＲＴＯＳなし・イベント駆動型アーキテクチャ（注１）が採用されており、メイン関数と２つの割り込みハンドラからなります。各々の動作を下記に示します。

メイン関数

周辺モジュール（割り込み要因）の初期化と割り込みコントローラを初期化した後、無限ループをしながら割り込みが入るのを待ち続けます。

タイマ割り込みハンドラ１ (ISR1: 低優先度割り込み)

・timer.c に記述されています
・周期タイマ（ＣＭＴ０）を使用し、ＬＥＤ１の点滅制御を行います
・500mesc 毎に点滅を繰り返します

タイア割り込みハンドラ２ (ISR2: 中優先度割り込み)

・timer.c に記述されています
・周期タイマ（ＣＭＴ１）を使用し、ＬＥＤ２の点滅制御を行います
・一度点灯させる毎に、消灯間隔（1000msec 単位）で増加していきます。
例　消・点・消・消・点・消・消・消・点 ･･･・ 消×６以上になると最初の間隔に戻ります

リアル通信割り込みハンドラ（ISR3：高優先度割り込み）

・sci.c に記述されています
・シリアルインターフェース（SCIF）を使用しエコーバックを行います。
・SCIF から受信した値をそのまま SCIF へ送信します。

これらの動作に、ＦＷが提供するマルチコア用プロセスペアの動作が加わり、最終的なサンプル全体の動作となります。本サンプルでは、各割り込みの開始と終了時にＷＤＴの停止と開始処理を行っています（注２）。各割り込みハンドラが仮想の専用ＷＤＴを持つ動作となります。動作イメージを図 2.2 に示します。またチェックポインティングはメイン関数行っており（注２）、その動作イメージは図 2.3 のようになります。

※注１　６章参照
※注２　４章参照

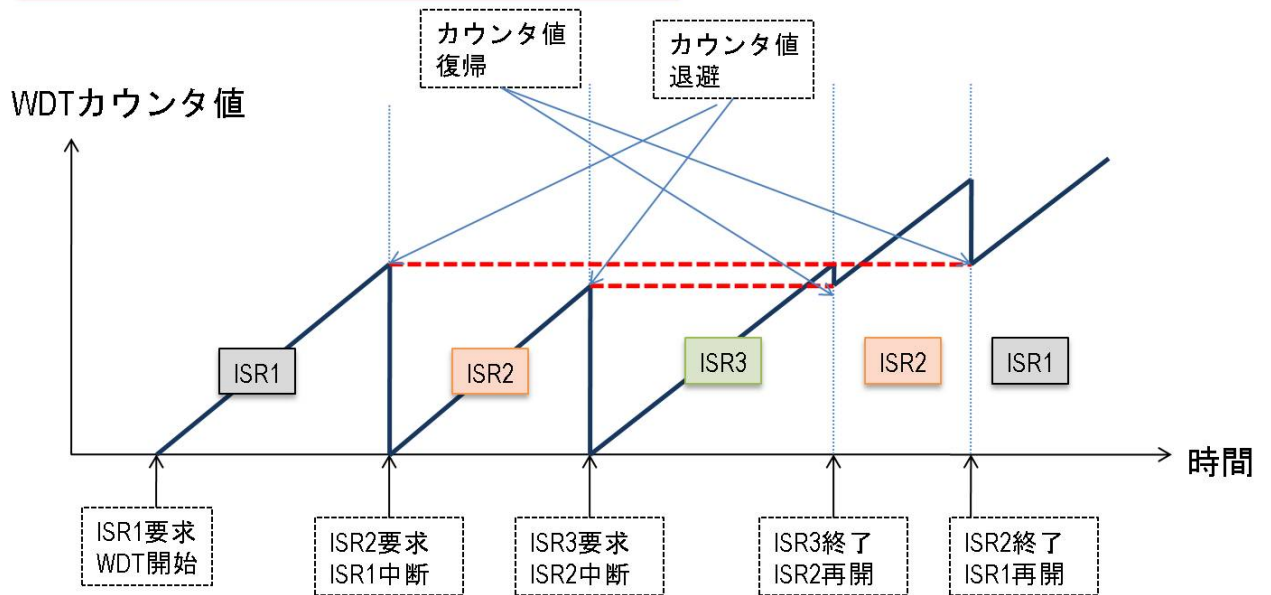



図 2.2：ＷＤＴの稼働イメージ

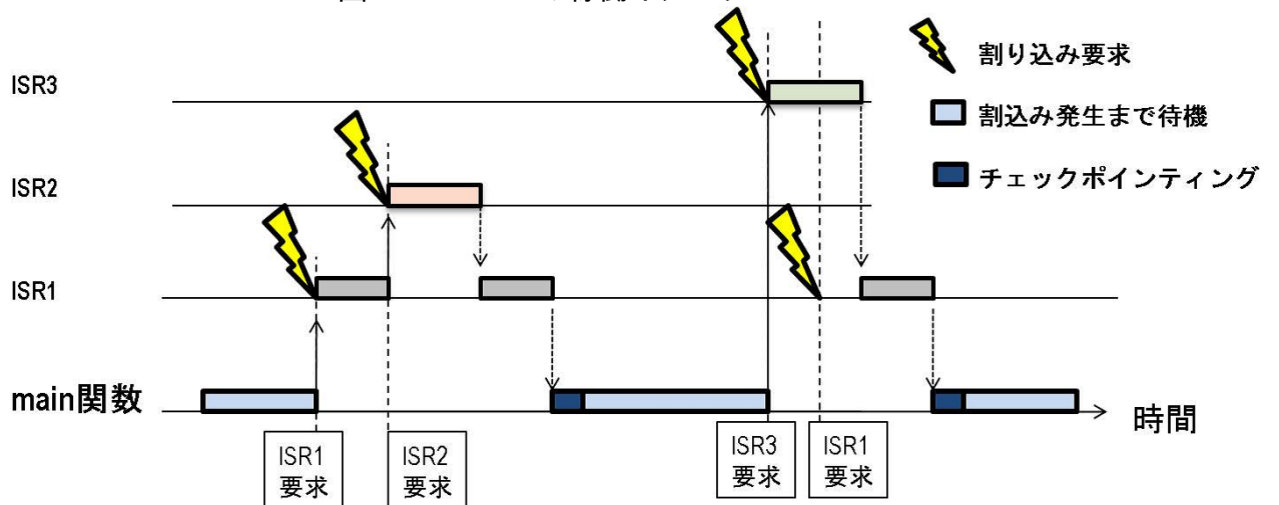



図 2.3：チェックポインティングの動作イメージ

第３章　適用手順

本節では、ＦＷに付属してあるサンプルプログラムを例に、ＦＷの適用手順について説明します。ＦＷの適用手順の大別を下記に示します。なお説明の為に、各手順に番号が付けてありますが、順不同ならびに場合によっては行わなくてもよい手順もあります。なお、全てのソースファイルは、ＣＰＵ＃０用、＃１用の２種類作成してください。どちらもほぼ同一の内容になります。差分となる点も後述してあるので参考にしてください。

適用手順

1：ＣＰＵ初期化処理の記述
2：チェックポイントタイミングの選択（ＦＷ or ＡＰＰ）
2.1 デフォルト設定の場合
2.2 デフォルト設定を解除した場合
3：ハンドラファイルの作成
3.1 テンプレートファイルの複製と修正
3.2 ベクタテーブルへの登録
3.3 チェックポイント情報の登録
3.4 チェックポイント関数の登録

１：ＣＰＵ初期化処理の記述

　ＣＰＵ及び周辺モジュール等ハードウェアの初期化を行ってください。”AP_SetUp.c”の関数名 ”AP_SetUp_bpard() ”にＣＰＵ及び周辺モジュール等、ハードウェアの初期化ルーチンのサンプルプログラムが記述してあります。必要に応じて、変更・追記を行ってください。

AP_SetUp.c

```
void AP_SetUp_board(){
サンプル記述
}
void AP_SetUp_IH(){
            :
}
```

２：チェックポイントタイミングの選択

デフォルトの場合、チェックポインティングはメイン関数が行うようになっています。メイン関数で別の処理を行わせたい場合や、メイン関数以外のＡＰＰで行いたい場合は、次の手順に沿ってそれぞれ作業を行ってください。

ファイル名“ `FW_config.h` ”内に“ `#define _FW_CKP_OFF` ”の１行をコメントアウトする事で、デフォルト設定を解除できます。

- デフォルト設定の場合（メイン関数でチェックポインティングを行う場合）

  → コメントアウトしない
  → 作業 2.1 へ

- デフォルト設定を解除する場合（ＡＰＰでチェックポインティングを行う場合）

  → コメントアウトする
  → 作業 2.2 へ

FW_config.h

```
// #define  _FW_CKP_OFF
           :
```

2.1　デフォルト設定の場合

デフォルト設定の場合、ＦＷの処理シーケンスは次のようになります

- 関数　”AP_SetUp_board() ”の呼び出し
　　↓
- **関数**　”AP_SetUp_IH() ”の呼び出し
　　↓
- 関数　”FW_main() ”の呼び出し
（以後、無限ループをしながら割り込み要求を待ち続け、割り込みハンドラの処理が無事に終わるとチェックポインティングを行います）

ＡＰＰで使用する周辺モジュール（割り込み要因）や割り込みコントローラの設定を行う必要があります。ファイル名“ AP_SetUp.c ”の関数“ AP_SetUp_IH() ”に記述してください。このとき、割り込みコントローラの設定には注意が必要です。ＣＰＵ＃０側のソースファイルでは、全ての割り込みをＣＰＵ＃０が受けつけるように、割り込みコントローラを設定してください。同様に＃１側では＃１が受けつける設定を記述してください。

本設定の場合、チェックポインティングは自動で行われます。しかし、任意の位置に“ FW_recover_checkpoint() ”と記述する事でチェックポインティングを行うことができます※。必要に応じて、複数の位置に記述する事も可能です。ＡＰＰの構造に合わせて行ってください。チェックポインティングの位置や回数に依らず、最後に行ったチェックポインティング時のチェックポイント情報のみが保存されます。

（※ ”FW_recover.h ”がインクルードされている必要があります。）

CPU#0側の設定例
(#1側はCPU#1で受けつけるように設定)
AP_SetUp.c
void AP_SetUp_board(){
:
}
void AP_SetUp_IH(){
:
}
/* CMTの設定*/
:
/*割込みコントローラの設定*/
/* CMT割込みをCPU#0で受けつける設定*/
INTC.IDCNT22.WORD = 0x4100;
:
/* CMT カウント開始*/
CMT.CMSTR01.WORD |= 0x0001;
CPU#1側の設定例
(#1側はCPU#1で受けつけるように設定)
AP_SetUp.c
void AP_SetUp_board(){
:
}
void AP_SetUp_IH(){
:
}
/* CMTの設定*/
(#0側と同じ内容でＯＫ)
/*割込みコントローラの設定*/
/* CMT割込みをCPU#1で受けつける設定*/
INTC.IDCNT22.WORD = 0x4300;
:
/* CMT カウント開始*/
CMT.CMSTR01.WORD |= 0x0001;

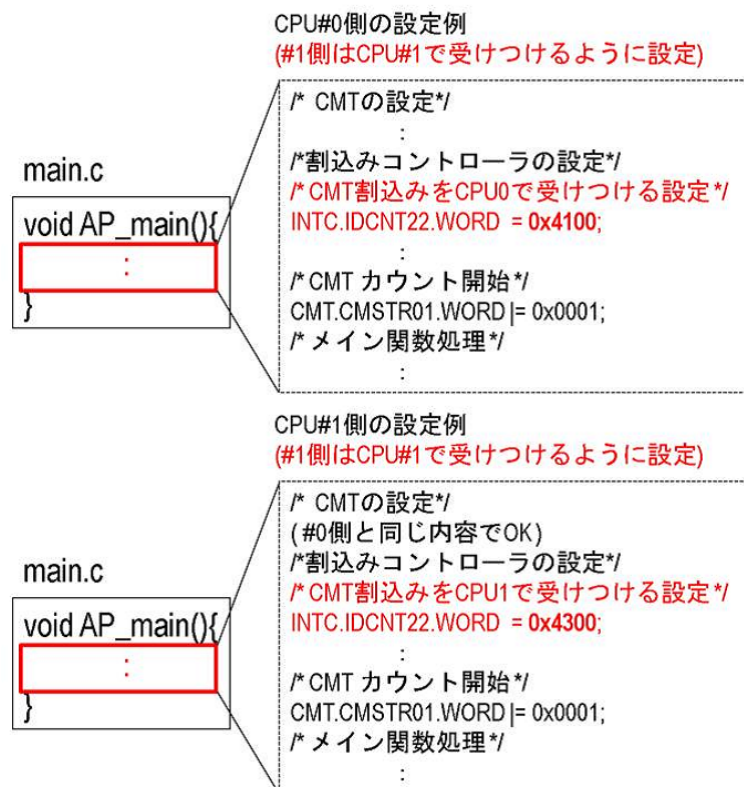
2.2　デフォルト設定を解除した場合

本節の場合、ＦＷの処理シーケンスは次のようになります

- 関数 ”AP_SetUp_board() ”の呼び出し
  ↓
- 関数 ”AP_main() ”の呼び出し

ＡＰＰのメイン関数となる“ AP_main() ”を作成してください。サンプルプログラムでは“ main.c ”に記述してあります。関数内では、メイン処理の他、ＡＰＰで使用する周辺モジュール（割り込み要因）や割り込みコントローラの設定も記述してください。このとき、割り込みコントローラの設定には注意が必要です。ＣＰＵ＃０側のソースファイルでは、全ての割り込みをＣＰＵ＃０が受けつけるように、割り込みコントローラを設定してください。同様に＃１側では＃１が受けつける設定を記述してください。

本設定の場合、チェックポインティングは自動で行われません。任意の位置に“ FW_recover_checkpoint() ”と記述する事でチェックポインティングを行うことができます※。必要に応じて、複数の位置に記述する事も可能です。ＡＰＰの構造に合わせて行ってください。チェックポインティングの位置や回数に依らず、最後に行ったチェックポインティング時のチェックポイント情報のみが保存されます。

（※ ”FW_recover.h ”がインクルードされている必要があります。）

３：ハンドラファイルの作成

この章では割り込みハンドラの記述方法および、チェックポイント情報に関連する設定、記述について説明します。

3.1　テンプレートファイルの複製と修正

割り込みハンドラを記述するにあたり、まずは“ IH_Template ”というファイルを複製し、任意のファイル名に変更し、ハンドラファイルを作成してください。サンプルプログラムの場合は ”sci.c ”と“ timer.c ”がハンドラファイルとなっています。複製後、次の２つの作業を行ってください。

- 複製して作成したハンドラファイル内の関数名“ ¡handler¿ ”の部分を任意のハンドラ名に変更してください。ハンドラ処理はこの関数内に記述してください。関数内の始めと終わりにＷＤＴの開始と停止関数の呼び出し記述があります。必要に応じて、削除・位置の変更・追加が可能です（詳細は後述のＷＤＴの使用法を参照のこと）。
- 複製して作成したハンドラファイル内の関数名“ <ckp_funk> ”の部分を任意の関数名に変更してください。この関数が後述 (3.4) のチェックポイント関数となります。

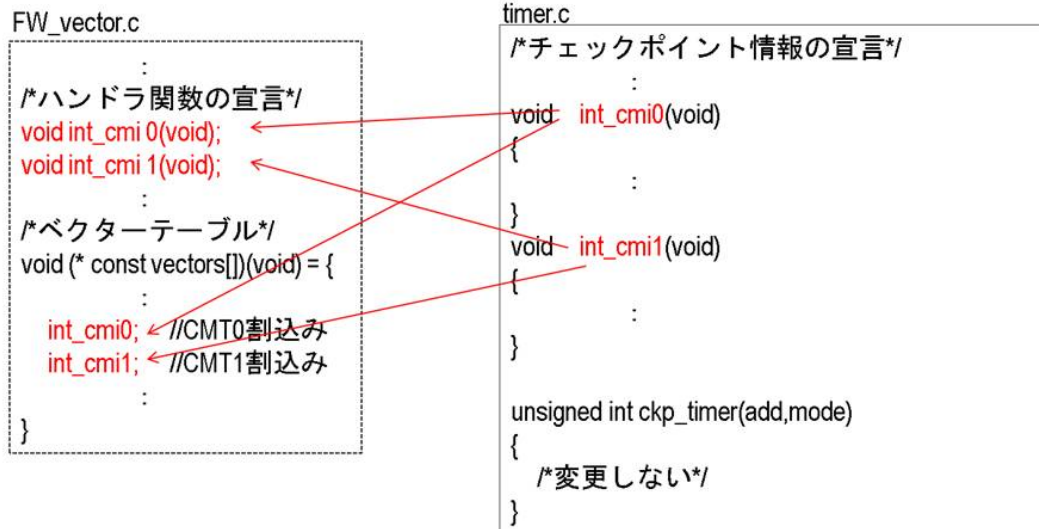
ひとつのハンドラファイル内に複数のハンドラ関数が存在していても構いませんが、チェックポイント関数は一つのファイルにつき、一つだけです。

IH_Template → timer.c

```
/*チェックポイント情報の宣言*/
                :
void <handler>(void) → int_cmi0
{
   FW_wdt_start();
   /*タイマハンドラ１処理記述*/
   FW_wdt_stop();
}

void <handler>(void) → int_cmi1
{
   FW_wdt_start();
   /*タイマハンドラ２記述*/
   FW_wdt_stop();
}
unsigned int <ckp_func>(add,mode)    →ckp_timer
{
   /*変更しない*/
}
```

3.2　ベクタテーブルへの登録

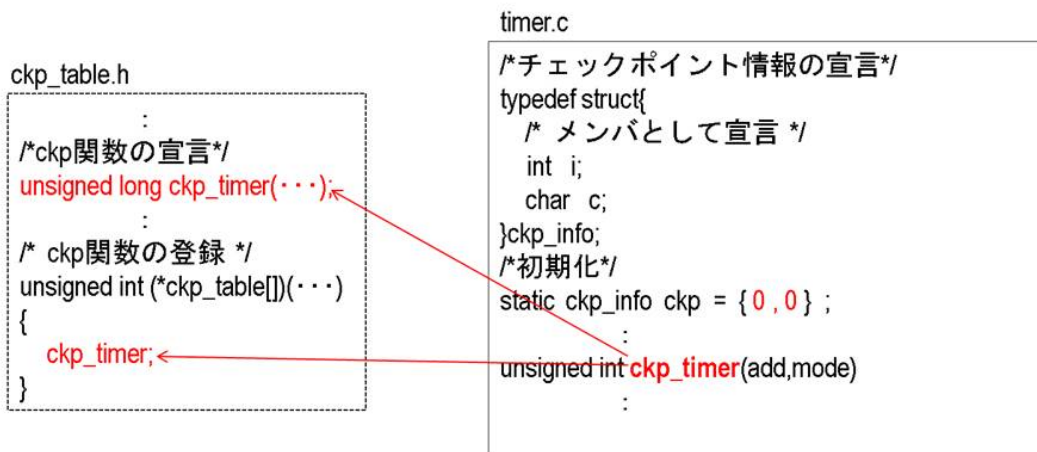
“ FW_vector.c ”に作成したハンドラ関数の宣言と、関数ポインタ配列への登録を行ってください。

3.3　チェックポイント情報の登録

下記の手順に沿って、チェックポイント情報（変数）を登録しておくことで、チェックポインティングが行われた際に、その時点でのチェックポイント情報が保存されます。また、何かしらの障害が発生し、システム回復が行われた際は、保存されたチェックポイント情報が、サブＣＰＵ側に引き継がれます。本節ではチェックポイント情報の登録方法および、登録したチェックポイント情報の使用法について説明します。

チェックポイント情報の登録は、作業 3.1 で作成したハンドラファイル内の構造体のメンバとして宣言します。その後、構造体の初期化を行ってください。変数名は任意で構いませんが、構造体名は変更しないでください。また、ファイル内でその変数を使用する際は、“ ckp. 変数名 ”という形で使用してください。チェックポイント情報の適用範囲はファイル単位となっています。したがって、チェックポイント情報が存在するファイルごとに登録作業を行う必要があります。チェックポインティングがされるときは、全てのチェックポイント情報が一括して保存されます。本節の作業はＣＰＵ＃０側、＃１側ともに同じ宣言順、変数名となるようにしてください。両ＣＰＵで整合性が取れなかった場合システム回復が正しく行われない可能性があります。

3.4　チェックポイント関数の登録

“ ckp_table.h ”に作業3.1で作成したチェックポイント関数の宣言と、関数ポインタ配列への登録を行ってください。“ ckp_table.h ”の内容はＣＰＵ＃０側、＃１側で同じ内容、記述順としてください。

※本作業を自動で行う、Ｒｕｂｙスクリプトが付属されています。Ｒｕｂｙを実行可能な環境であれば、作業3.3まで終了後、“ make_ckp_table.rb ”を実行すれば、本節の作業を行う必要はありません。

第4章　アプリケーションごとに個別指定可能な機能

4.1　ＷＤＴについて

エラー検知機能として、本ＦＷでデフォルトとしているＷＤＴですが、アプリケーションに応じて次の事が可能です。

4.1.1　任意箇所でのＷＤＴの操作

ＷＤＴの操作に関するコンポーネントとして次の関数が用意してあります。いずれの関数も“ FW_wdt.h ”をインクルードすれば任意のプログラム内から呼び出し可能です。ＷＤＴの開始と停止関数はペアでの使用を推奨します。すなわち、ある関数やファイル内でＷＤＴを開始させたら、同関数もしくはファイル内で停止処理を行ってください。いずれの関数も、不必要に乱用した場合、正しくエラー検知を行えない可能性があるので注意が必要です。

関数名	説明
FW_wdt_start(void)	ＷＤＴを開始する
FW_wdt_stop(void)	ＷＤＴを停止する
FW_wdt_reset(void)	ＷＤＴカウント値をリセットする
FW_wdt_cntset(int)	ＷＤＴのカウント値を引数により指定する ※引数０～２５５

4.1.2　ＷＤＴの使用法の選択

“ FW_config.h ”内の ” #define　_FW_WDT_OFF ”のコメント指定を外すことで、ＷＤＴをＦＷのエラー検知機能としてではなく、ＡＰＰに応じた使用が可能です。この場合ＷＤＴの初期化はＡＰＰで行ってください。またベクタテーブルも変更する必要があります。エラー検知もＡＰＰで用意する必要があります。典型的なエラー検知手法のひとつである受入れテストなど、ＡＰＰに応じたエラー検知機能を実装してください。その際にＣＰＵを切り替える方法については 4.3 で説明します。

4.2　チェックポインティングに関して

“ FW_recover.h ”をインクルードし、関数 ”FW_recover_checkpoint() ”を呼び出す事で、任意の位置でチェックポインティングを行えます。使用箇所、使用回数に制限はありませんが、エラー検知のタイミングと、アプリケーションの構造に依っては正しくシステム回復が行えなくなるので、よく検討し、適切な位置でチェックポインティングを行ってください。

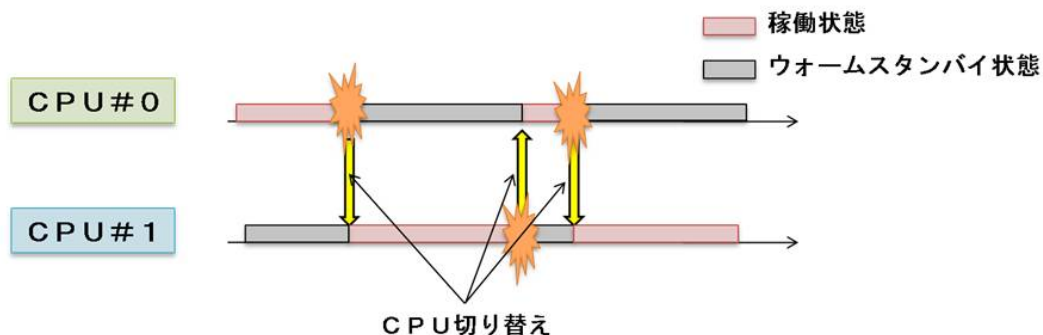
4.3　ＣＰＵ切替機能について

エラー検知をＷＤＴを使用し方法ではなく、ＡＰＰ独自で用意する場合にＣＰＵ切替を行う方法について説明します。ＣＰＵの切替は“ FW_recover_core_switch() ”という関数を呼び出すことで利用可能です。利用例を図 4.1 に示します。エラー検知を行うファイルで ”FW_recover.h ”をインクルードし、切替関数を切替関数を応用してください。

```
if(エラー判定が真ならば・・・) {
        FW_recover_core_switch();
}
```

図 4.1：利用例

4.4　ＣＰＵ多重切替機能について

ＣＰＵ多重切替機能を使用する事も可能です。ＣＰＵ多重切替とは下図のような動作をします。

この機能を使用するには、まず“ FW_config.h ”内の“ #define　_FW_M_SW_OFF ”をコメントアウトしてください。次に“ AP_SetUp.c ”内の“ AP_SW_reset() ”という関数内にソフトウェアリセットを行う記述が必要となります。ソフトウェアリセットサンプルプログラムが記述してあるのでそのまま利用する事も可能です。

第5章　マルチコア用リカバリブロックへの発展的応用

本章では、本ＦＷのＣＰＵ切替機能を応用した、マルチコアを使用したリカバリブロックについて説明します。マルチコアでは同じソフトウェアを異なるＣＰＵに実装するものですが、リカバリブロックの場合は異なる設計のソフトウェアを用意します。プロセスペアと違いマルチＣＰＵ環境は必須ではありませんが、マルチコアと、本ＦＷのＣＰＵ切替機能を活用する事で、より手軽に導入することも可能であると思われます。ＣＰＵ＃０、＃１に、異なる設計に基づくアプリケーションを実装すれば、必然的にＣＰＵが切り替わった際は、別バージョンのソフトウェアが稼働する事になります。この時注意しなければならないのは、チェックポイント情報の扱いです（3章の手順3.3参照)。

第6章　注意事項

6.1　対象ソフトウェア構造について

本ＦＷのサンプルプログラムではイベント駆動型のアーキテクチャを採用していますが、これに限定されるわけではありません。同様に、ＲＴＯＳなしの構造である、時間駆動型アーキテクチャを採用したアプリケーションに適用する事も可能です。ただし、ＲＴＯＳベースのアーキテクチャに関しては動作確認を行っていませんので、その動作を保証する事はできません

6.2　対象フォルトモデル

本ＦＷのコンセプトの一つとして、『エラーやアプリケーションに依らず、ある程度汎用的に適用可能であること』という点があります。そのためアプリケーションへの制限は少なく、対象フォルトモデルは明確に定義してありません。応用範囲が広い半面、最終的に対処可能なフォルトモデルはアプリケーションに依存するものとなり、ＦＷの機能を十分理解し活用されることが望ましいです。

6.3　開発環境ＨＥＷに関する点

本ＦＷは統合開発環境ＨＥＷを用いて開発されています。ＨＥＷ以外で本ＦＷを利用する場合は下記の点に注意してください。

(1) コンパイラによる最適化は行っていません

(2) ＨＥＷのコンパイラオプションでマクロ定義を行っています (”ＲＯＭ”を定義しています)

(3) ＨＥＷのリンカオプションでスタック (シンボル名：stack) を定義し、使用しています

(3) についてはＨＥＷを用いる場合は、スタックの開始番地はリンカオプションで変更する事が可能です。また、アプリケーションプログラム内で別に定義し、使用しても構いません。